ADTs and Recursion!

THURSDAY, APRIL 17

Section materials curated by Jonathan Coronado, Yasmine Alonso, and Sean Szumlanski, drawing upon materials from previous quarters.

This week's section exercises explore Grids, Maps, Sets, and Stacks! By the end of this week you'll be well-versed in all kinds of ADTs, which we'll learn about in class, and know the best use cases for each of them.

Remember that every week we will also be releasing a Qt Creator project containing starter code and testing infrastructure for that week's section problems. When a problem name is followed by the name of a **.cpp** file, that means you can practice writing the code for that problem in the named file of the Qt Creator project. Here is the zip of the section starter code:

📦 Starter project

1. Grid Basics (**grid.cpp**)

Topic: Grids

a. Maximum of a Row in a Grid

Write a function named **maxRow** that takes a grid of non-negative integers (numbers from 0 to infinity) and an in-bounds grid location and returns the maximum value in the row of that grid location.

Solution

```
// solution1 (manally loop through the row in the grid)
int maxRow(Grid<int>& grid, GridLocation loc) {
    int max = -1;
    for (int col = 0; col < grid.numCols(); col ++) {
        if (grid[loc.row][col] > max) {
            max = grid[loc.row][col];
        }
    }
    return max;
}
```

```
// solution2(use GridLocationRange)
int maxRow(Grid<int>& grid, GridLocation loc) {
    int max = -1;
    int endCol = grid.numCols() - 1;
    for (GridLocation cell : GridLocationRange(loc.row, 0, loc.row,
endCol)) {
        if (grid[cell] > max) {
            max = grid[cell];
        }
    }
    return max;
}
```

b. Average value

Write a function named **avgNeighborhood** that takes a grid and a grid location and returns the average of all the values in the neighborhood of the grid location. A neighborhood is defined as all cells in a grid that border the grid location in all four directions(N, S, E, W). If the average is

an integer, return a truncated average.

Solution

```
// solution1 (we put the 4 locations in a Vector and loop over them)
int avgNeighborhood(Grid<int>& grid, GridLocation loc) {
    Vector<GridLocation> possibleLocations = {
        {loc.row - 1, loc.col}, // north
        {loc.row + 1, loc.col}, // south
        {loc.row, loc.col + 1}, // east
        {loc.row, loc.col - 1}  // west
    };

    int sum = 0;
    int numValidLocations = 0;
    for (GridLocation dir : possibleLocations) {
        if (grid.inBounds(dir)) {
            sum += grid[dir];
            numValidLocations += 1;
        }
    }
    return sum / numValidLocations;
}

// solution2 (Don't do this please!! We manually get all 4 locations
and sum them up)
int avgNeighborhood(Grid<int>& grid, GridLocation loc) {
    int sum = 0;
    int numValidLocations = 0;

    GridLocation north {loc.row - 1, loc.col};
    if (grid.inBounds(north)) {
        sum += grid[north];
        numValidLocations += 1;
    }

    GridLocation south {loc.row + 1, loc.col};
    if (grid.inBounds(south)) {
        sum += grid[south];
        numValidLocations += 1;
    }

    GridLocation east {loc.row, loc.col + 1};
    if (grid.inBounds(east)) {
        sum += grid[east];
        numValidLocations += 1;
    }

    GridLocation west {loc.row, loc.col - 1};
    if (grid.inBounds(west)) {
        sum += grid[west];
        numValidLocations += 1;
    }

    return sum / numValidLocations;
}
```

2. Friends (**friendlist.cpp**)

a. Building the friendList

Write a function named **`friendList`** that takes in a file name, reads friend relationships from the file, and writes them to a **`Map`**. **`friendList`** should return the populated **`Map`**. Friendships are bi-directional, so if Abby is friends with Barney, Barney is friends with Abby. The file contains one friendship relationship per line, with names separated by a single space. You do not have to worry about malformed entries.

If an input file named **`buddies.txt`** looked like this:

```
Barney Abby
Abby Clyde
```

Then the call of **`friendList("buddies.txt")`** should return a resulting **`map`** that looks like this:

`{"Abby":{"Barney", "Clyde"}, "Barney":{"Abby"}, "Clyde":{"Abby"}}`

Here is the function prototype you should implement:

`Map<string, Set<string> > friendList(String filename)`

Solution

```
Map<string, Set<string> > friendList(string filename) {
    ifstream in;
    Vector<string> lines;

    if (openFile(in, filename)) {
        lines = readLines(in);
    }

    Map<string, Set<string> > friends;
    for (string line: lines) {
        Vector<string> people = stringSplit(line, " ");
        string s1 = people[0];
        string s2 = people[1];
        friends[s1] += s2;
        friends[s2] += s1;
    }
    return friends;
}
```

b. Finding common friends

Write a function named **`mutualFriends`** that takes in the friendList above, and two strings representing two friends, and returns the names of the mutual friends they have in common. For example, if the friendList is **`{"Abby":{"Barney", "Clyde"}, "Barney":{"Abby"}, "Clyde":{"Abby"}}`** and friend1 is **`Barney`** and friend2 is **`Clyde`**, then your function should return **`{"Abby"}`**

Solution

```
Set<string> mutualFriends(Map<string, Set<string> >& friendList, string
friend1, string friend2) {
    return friendList[friend1] * friendList[friend2];
}
```

3. Twice (`twice.cpp`)

Topic: Sets

Write a function named **`twice`** that takes a vector of integers and returns a set containing all the numbers in the vector that appear exactly twice.

Example: passing **`{1, 3, 1, 4, 3, 7, -2, 0, 7, -2, -2, 1}`** returns **`{3, 7}`**.

Bonus: do the same thing, but you are not allowed to declare any kind of data structure other than sets.

Solution

```
// solution
Set<int> twice(Vector<int>& v) {
    Map<int, int> counts;
    for (int i : v) {
        counts[i]++;
    }
    Set<int> twice;
    for (int i : counts) {
        if (counts[i] == 2) {
            twice += i;
        }
    }
    return twice;
}

// bonus
Set<int> twice(Vector<int>& v) {
    Set<int> once;
    Set<int> twice;
    Set<int> more;
    for (int i : v) {
        if (once.contains(i)) {
            once.remove(i);
            twice.add(i);
        } else if (twice.contains(i)) {
            twice.remove(i);
            more.add(i);
        } else if (!more.contains(i)) {
            once.add(i);
        }
    }
    return twice;
}
```

4. Check Balance (`balance.cpp`)

Topic: Stacks

Write a function named **`checkBalance`** that accepts a string of source code and uses a **`Stack`** to check whether the braces/parentheses are balanced. Every **`(`** or **`{`** must be closed by a **`}`** or **`)`** in the opposite order. Return the index at which an imbalance occurs, or -1 if the string is balanced. If any **`(`** or **`{`** are never closed, return the string's length.

Here are some example calls:

```
//   index    01234567890123456789012345678
checkBalance("if (a(4) > 9) { foo(a(2)); }")
// returns -1 (balanced)

//   index    012345678901234567890123456789012
checkBalance("for (i=0;i<a;(3};i++) { foo(); )")
// returns 15 because } is out of order

//   index    0123456789012345678901234
checkBalance("while (true) foo(); }{ ()")
// returns 20 because } doesn't match any {

//   index    01234567
checkBalance("if (x) {")
// returns 8 because { is never closed
```

Solution

```
int checkBalance(string code) {
    Stack<char> parens;
    for (int i = 0; i < code.length(); i++) {
        char c = code[i];
        if (c == '(' || c == '{') {
            parens.push(c);
        } else if (c == ')' || c == '}') {
            if (parens.isEmpty()) {
                return i;
            }
            char top = parens.pop();
            if ((top == '(' && c != ')') || (top == '{' && c != '}')) {
                return i;
            }
        }
    }

    if (parens.isEmpty()) {
        return -1; // balanced
    }
    return code.length();
}
```

5. Count Escape Routes (**escape.cpp**)

Topics: Recursive counting

Given an input maze and a start GridLocation, write a recursive function that counts the number of ways we can escape the maze. In this maze, we can only take one unit steps in two directions: south and east. Just like the maze in assignment 2, the only exit out of the maze is at the bottom right corner.

Solution

```
int countWaysToEscape(Grid<bool>& maze, GridLocation location) {
    // if we are out of bounds or this location is blocked, stop
searching
    if (!maze.inBounds(location) || !maze[location]) {
        return 0;
    }

    // if we are the end of the maze, we've found one way to escape
    if (location == GridLocation{maze.numRows() - 1, maze.numCols() -
1}) {
        return 1;
    }

    int waysSouth = countWaysToEscape(maze, {location.row + 1,
location.col});
    int waysEast = countWaysToEscape(maze, {location.row, location.col
+ 1});

    return waysSouth + waysEast;
}
```

---

6. Random Shuffling (**`shuffle.cpp`**)

How might the computer shuffle a deck of cards? This problem is a bit more complex than it might seem, and while it's easy to come up with algorithms that randomize the order of the cards, only a few algorithms will do so in a way that ends up generating a uniformly-random reordering of the cards.

One simple algorithm for shuffling a deck of cards is based on the following idea:

- Choose a random card from the deck and remove it.
- Shuffle the rest of the deck.
- Place the randomly-chosen card on top of the deck. Assuming that we choose the card that we put on top uniformly at random from the deck, this ends up producing a random shuffle of the deck.

Write a function

```
string randomShuffle(string input)
```

that accepts as input a string, then returns a random permutation of the characters of the string using the above algorithm. Your algorithm should be recursive and not use any loops (**`for`**, **`while`**, etc.).

The header file **`"random.h"`** includes a function

```
int randomInteger(int low, int high);
```

that takes as input a pair of integers low and high, then returns an integer greater than or equal to low and less than or equal to high. Feel free to use that here.

Interesting note: This shuffling algorithm is a variant of the Fisher-Yates Shuffle. To learn how to prove it works correctly, take CS109!

Solution

Here is one possible solution:

```
string randomShuffle(string input) {
    /* Base case: There is only one possible permutation of a string
    * with no characters in it.
    */
    if (input.empty()) {
        return input;
    } else {
        /* Choose a random index in the string. */
        int i = randomInteger(0, input.length() - 1);
        /* shuffle the rest of the string, add the removed character on
the front
        * the string.
        */
        return input[i] + randomShuffle(input.substr(0, i) +
input.substr(i + 1));
    }
}
```

This function is based on the recursive observation that there is only one possible random shuffle of the empty string (namely, itself), and then using the algorithm specified in the handout for the recursive step.